A Behavior Feature Generation Method for Obfuscated Malware Detection

Rui Wang
State Key Laboratory of Information Security
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
wangrui@iie.ac.cn

Xiaoqi Jia
State Key Laboratory of Information Security
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
jiaxiaoqi@iie.ac.cn

Chujiang Nie
State Key Laboratory of Information Security
Institute of Information Engineering, Chinese Academy of Sciences
Beijing, China
niecj@is.iscas.ac.cn

Abstract—**Detection based on features is most popular way to prevent malware these days. Current feature abstracting and matching methods are susceptible to obfuscation techniques, and cannot deal with the variants which are emerging quickly. This paper proposes a malware feature extracting method based on its behaviors. This method can abstract the critical behaviors of malware and the dependencies between them through dynamic analysis, and generate the features to defeat malware obfuscations considering semantic irrelevancy and semantic equivalency to improve the describing capabilities of the malware features. This paper also designs a corresponding detecting method based on these features. The experiment results show that our method is more resilient to malware obfuscation techniques, especially for real world malware variants.**

Keywords-malware, feature exstracting, dynamic taint analysis, behavior dependency, semantic analysis

I. INTRODUCTION

Malware is one of the most serious challenges to computer and Internet today. Obfuscation using by malware is a popular method to avoid malware detecting. The traditional malware detection abstracts syntactic features from malware samples to distinguish the malware, and usually require accurately match. The syntactic features should be frequently updated and cannot manipulate simply obfuscating. Although the obfuscated malware omits its syntactic features, it is very possibly that the semantic remains. Then semantic-based detection has been used in some previous works, such as Christodorescu etc. [3] developed a robust way which abstracted features by semantic analysis, but performed poor against instruction-replacing for preventing some obfuscating methods.

The capability and efficiency of feature-based malware detection depends on the description capability of features. Recently, the bottleneck of feature-based detection is the gap between the malware's features and its real behavior [1]. Present methods based on sequence and control flow-graph cannot describe the internal logic of malware, while the dependence graph which focuses on data dependence with little control dependence cannot give a well expression to the behavior logic. How to abstract the essential characteristics of malware to prevent from obfuscation, thus to make malware detecting more precisely and efficiently, is the main focus of malware prevention these days.

We present a prototype system based on behavior semantic to abstract the character of malware. Our implementation begin with analyzing the malware samples through dynamic taint analysis, abstracting the critical behaviors, the data dependence and control dependence, then reconstruct behavior logic of malware and execute semantic analysis. The characters collected with our method can be adapted to detect malwares variants derived by obfuscation, perform better on malwares variants, and reduce the delay between the malware variants arising and features updating.

The main contributions of this paper are as follows:

- We extend presently DDG (Data Dependence Graph) and CDG (Control Dependence Graph), and bring out a novel method to construct Dependence Graph.
- We propose a semantic analysis and process method for behavior characters, which focuses on semantic-irrelevant and semantic-equivalent to improve the description capability of features.
- We implement a prototype abstracting features, and evaluate the features by constructing corresponding detecting arithmetic.

This paper is organized as follows: section 2 is system overview; section 3 gives a brief description of the malware analysis on behaviors; section 4 introduces the semantic analysis. Our algorithm based on the abstracted features is presented in section 5. The evaluation details are shown in section 6. We introduce relative work in section 7, and give a conclusion in section 8.

II. SYSTEM OVERVIEW

We implement a prototype to abstract the behavior semantic features of malware. Using dynamic analysis, this prototype abstracts malware's critical behaviors and the dependency of these behaviors which can be concluded as behavior semantic features to detect polymorphic malware. This paper focuses on improving the capabilities of malware feature to describe malware behavior and prevent obfuscating.

Our prototype system consists of behavior analysis module, semantic analysis module and detection verified module, as is illustrated in Fig.1.

Behavior analysis module is composed of analysis environment, behavior analyzer and analysis output. Our prototype is implemented on WooKon platform, which constructed on a hardware emulator Qemu [8]. We run a Windows XP on WooKon as the environment of malware. This architecture is highly transparent for malware, because of the separated running environments of analysis module and analysis object. We enhance present dynamic taint technology with supporting reverse analysis to trace and analyze malware in fine grain.

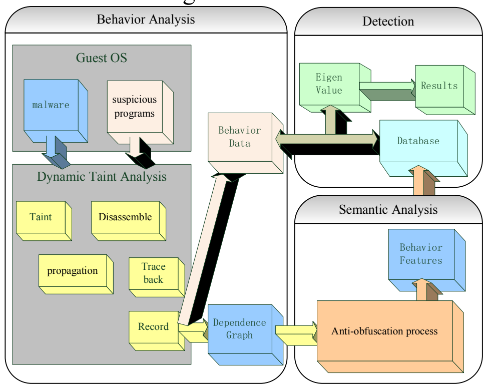

Figure 1. Prototype System Overview.

The output of this module is behavior dependence graph, which is composed of system calls and the dependencies of them. Almost all the operations on sensitive data structures are relative to critical system calls [1, 2]. So we use critical system calls to describe critical behaviors of malware. To present the logic relation between malware behaviors, we use both the data dependence and control dependences to construct dependence graph.

To prevent from obfuscating, we abstract behavior characteristics from dependence graph through semantic analysis. Usually, the key behaviors of obfuscated malware, namely the semantic of statements in the main function, keep stable. So we implement the semantic analysis with the purpose of recognizing the obfuscated instance of existent samples. To manipulate common obfuscating technologies, we process the dependence graph through both semantic irrelevancy and semantic equivalency. The outputting malware behavior feature graph perform obviously effect in prevent from common obfuscation.

To evaluate the abstracted behavior features, we design a detection verified module which is processed as follow. First we analyze the target program by dynamic taint analysis, reveal the system calls and dependence, and match the features of sample set. Second we set the nodes and edges with weight value according to the operating sensitivity, and set the detecting threshold. Last we calculate the eigen value on the matching result, and then detect malware when eigen value is larger than detecting threshold.

III. Behavior Analysis

The behavior analysis constructs behavior dependence graph on critical system calls and the data dependencies and control dependencies between them by analyzing the critical behavior flow when tracing the process of malware samples by dynamic taint analysis.

A. *Taint Analysis*

Our system analyzes the behaviors of malware by dynamic taint analysis. Through tracing and analyzing corresponding behaviors and data processing, our system can abstract critical system calls, the corresponding data dependencies and control dependencies, and instructions relative with tainted data.

We build a taint source system call list which contains four main parts naming file, network, register and process. These four parts is denoted as:

$F_{sens}=\{FileTaint, RegistryTaint, NetworkTaint, ProcessTaint\}$

When the system call belonging to F_{sens} occurs, the outputting parameters of it are marked as taint according to the type of taint source.

We construct shadow memory to save taint records as $Taint=\{Address, Length, Status, Type\}$, *Address* stands for the start address of taint, *Length* stands for the length of taint counted by byte, *Status* stands for the status of taint, *Type* stands for the type of taint iterated by general-purpose register, flag register and memory.

After taint having been marked, we analyze malware by single-stepping, and calculate taint propagation path on taint propagating rules. The taint propagating rules contain system call part and instruction part. The taint propagating rules of system calls is built for each function. Considering system call *F*, if the inputting parameters $Parameter_{in} \in Taint$, then the outputting parameters $Parameter_{out} \in Taint$. The taint propagating rules of instruction is built for each type, and we focus on memory operation instructions, operation instructions and control flow transfer instructions.

When taint propagating, if the taint source T_1 from system call F_1 extends to F_2 as input parameter, then F_2 data-depend on F_1. When control flow transfer instruction I_1 is to be executed and this instruction is influenced by the tainted EFLAGS, we disassemble the current instruction and the subsequent instruction, and then calculate the dominators to determine the control domain D_1, of I_1 [5], i.e., when the taint source of system call F_1 extends to I_1, all *F* in the domain D_1 control-depend on F_1.

Taint bleach when taint data have been rewritten by irrelevant data. We start reverse analysis to abstract the dependencies between behaviors when taint is bleaching.

B. *Trigger Condition Managing Engine*

Malware usually include some trigger condition, which is a judge condition in code running. Malware execute vicious behaviors only when the trigger conditions meet, which is used to protect itself.

We construct trigger condition managing engine which can recognize the trigger condition and try to meet it to continue the code executing. We focus on three main cases namely delaying by calling *sleep* function, cyclic algebra calculating and judging system time. For delaying by calling *sleep* function, we hook the *sleep* function in operation system, read the delay from stack when the sleep function being called, and modify the *cpu_get_ticks* series function by setting it return value with the delay to alter the hardware time to meet the trigger condition. The cyclic algebra calculating is not related to taint operations, then we firstly detect the loop in no-taint instructions and determine the instructions that put forwards the loop, secondly NOT the corresponding bit in EFLAGS to force the loop out at the next execution. For judging system time, we determine whether the control flow transfer instruction depends on the taint data from system time, and set the flag bit corresponding to this instruction to continue the executing flow.

C. Behavior Dependence Graph

In this section, we construct behavior dependence graph as the base of describing characters.We present behavior dependence graph G_i as:

$$G_i=\{N_E,N,C,D,Ins\}$$

N_E stands for the entry node, N stands for the node set, C stands for data dependence edge set, D stands for control dependence edge set, and *Ins* stands for the instructions related. It is possible that more than one G_i can be found during analyzing malware..

At the beginning of analysis, T_i is an empty set. Then we set the N_E of G_i with critical system call with tainted output, and refresh the shadow memory. Then the iterating procedure of computing the taint propagating process during every instruction in single-stepping will go on.

When a new system call occurs, the input parameters will be parsed, and a node will be added in G_i if the input parameters have been tainted. By the way, if a file has been copied to memory, the reading or writing behaviors should be transfer into corresponding system call node.

The dependence edge adding process is implemented by reversely computing taint propagation when adding new node, and can be divided into data dependence edge adding and control dependence edge adding. At the same time of adding new node, we query the taint data of current node and the dependencies between current node and the node which generating taint data, then add data dependence edges between current node and corresponding nodes.

About control dependence edge adding, if the current system call is within the control domain of some instruction, we add an edge between current node and the system call being taint source. To analyze control domain, we integrate our system with disassemble engine. When the control flow transfer occurs, we disassemble the following instructions recursively, and compute the control domain.

The constructing process may finish in two situation: 1)when all the taint data have been bleached; 2)when the execution of malware has completed.

IV. Semantic Analysis

The present variants of malware are usually generated by obfuscation technologies, which can transfer the signature to avoid the current detection method.

On the basis of behavior dependence graph of malware, we abstract the semantic features from vicious behavior, and manipulate semantic irrelevant call and semantic equivalent call well to prevent obfuscation.

We build anti-obfuscation engine in semantic domain to treat behavior obfuscation. By analyzing behavior obfuscation technologies, we classify it in two types namely semantic irrelevant system call and semantic equivalent system call. The former refer that malware usually contain some system calls that have nothing to do with its purpose. The latter refer to change the system call sequence by equivalent system call replacing and loop transfer.

For semantic irrelevant system call, we eliminate it during the process of taint propagating. The character of semantic irrelevant system call is that they don't change the system status. The set influencing system status can be denoted as *SChange*. The system call occurring on the path of the *Taint* propagating is denoted as N_1, and the taint extending path following it as L_N. When N_1 occurring, $L_N=\emptyset$; when the taint propagating to N_2, $L_N=\{N_1,N_2\}$. The condition of semantic irrelevant system call is $L_N=\emptyset$, or $L_N\neq\emptyset \wedge (\forall N\in L_N, N\notin SChange)$.

For semantic equivalent system call, we discuss it in two cases, namely equivalent system call replacing and loop transfer. Equivalent system call replacing is a great challenge in character abstracting and malware detecting. This paper recognizes the system calls by the set of semantic equivalent sequences, and adds set node in behavior dependence graph. We denote the equivalent sequence library as:

$$L_{equ}=\{S_1,S_2,\ldots,S_n\}$$

S standing for system call sequence. The set of semantic equivalent sequence library can be denoted as:

$$F=\{L_{equ1},L_{equ2},\ldots,L_{equn},Index\}$$

All S in L_{equ} have the same behavior and can be altered by each other. The *Index* in F stands for the index set $\{E_a,E_b,E_c,\ldots\}$, and E_i stands for the name of the first node of S_i which corresponding to a set which contains the same first node. The set node represents a semantic equivalent sequence library, and all the system call sequence in L_{equ} will be map as the same set node. If the sequence in behavior dependence graph belongs to a semantic equivalent sequence library, it should be reduced as set node.

Another case in semantic equivalent is loop translation. To prevent from the interference caused by equivalent loop translation, we reduce the loop into once execution by semantic methods. We consider the loop relative to taint data, and record the operation domain for every occurrence of taint system call. When there is a loop, we compare the address of the system call in loop. If the taint operating address in successor node equal to the sum of the taint operating address and the length of, the prior node, the nodes can merge, and set the range as the sum of the two ranges.

Through semantic analysis, we describe the features of malware as behavior feature graph:

$$G=\{N_E,N,N',C,D\}$$

In which, N_E, N, N', C, D stands for the entry node of graph, general node, set node, control dependence edge set and data dependence edge set respectively. The malware feature is represented as behavior dependence graph set:

$$T=\{G_1,G_2,...,G_n\},\ n\in N$$

Fig.2 illustrates the process of constructing behavior feature graph by semantic analysis, when analyzing sample Forbot. Having abstracted the behavior dependence graph (graph (a)), we first identify and eliminate the semantic irrelevant system calls, e.g., the sequence *CreateFile*, *ReadFile*, then reduce the loop of *ReadFile* and *SendTo*, at last, recognize sequence *CreateFile*, *ReadFile* belongs to semantic equivalent library *READ*, and replace this sequence by *READ*. The output of this process is behavior feature graph (graph (b)).

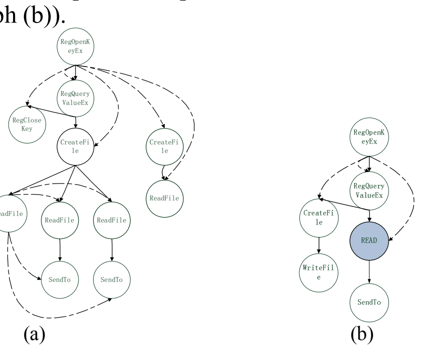

Figure 2. Forbot anti-obfuscation treatment.

V. MALWARE DETECTION

The detection based on behavior feature graph distinguish the critical system calls and dependencies of testing code, then match it to existing behavior features in feature library, and compute the matching score.

Count set $M_x=\{n_N,n_{N'},n_C,n_D\}$ is assigned to each T_x, to maintain the count of the nodes and edges which have been matched to T_x in detection. We trace the execution flow of object code, and match the behavior features with system calls and dependencies from taint analysis. The first step is matching the entry to the feature graph G_{xy} ($G_{xy}\in T_x$). Then we mark the return value of the entry system call as taint source and begin taint propagation. For general system call, the parameters of present system call should be determined whether refer to the taint data. If not, the system call will be omitted and taint propagate continuously; if the parameters refer to taint data, the system call will be matched to the node in behavior feature graph and the dependencies of it will matched to the edges.

The matching area of general nodes includes system call name and system call parameters. The name matching compares two system calls by string, and the parameters matching should compare the parameter type. We take four data type into considering, namely *handle*, *string*, *enumeration* and *struct*, and design special matching rules for different system call type.

There are some differences in set node case. When matching the set node, we step in semantic equivalent sequence library for advance matching. The matching process in semantic equivalent sequence library will end when a sequence has been matched, and then return to previous feature graph.

The edge matching goes at the same time of node matching. When the node is matching, the dependencies can be computed out by reverse analysis, and then is matched to the edges of feature graph.

The following step is calculating weighted eigen value by the result of matching. Since system call, data dependency and control dependency performs different influence on the behavior of malware, we set the weight value of general node, set node, control dependence edge and data control dependence as 0.5, 1, 0.2 and 0.5 respectively. The formula for calculating eigen value is expressed as follow:

$$E=\frac{\sum_{x=N,N',C,D} w_x n_x}{\sum_{x=N,N',C,D} {}_{Graph} w_x n_x}$$

VI. EXPERIMENTS

To verify the effectiveness of the feature of malware abstracted by our method, we chose some typical malware and corresponding variants for testing, and then analyzed and evaluated the result.

In this section, we chose NetSky and SdBot as examples to analyze the experiment results. The samples of NetSky were got from VX Heavens [10]. For SdBot, we generated SdBot.ma by manually obfuscaton and SdBot.up, SdBot.ex and SdBot.as by UPX, EXECryptor and Asprotect respectively, besides samples got from VX Heavens.

TABLE I. EIGEN VALUE OF NETSKY VARIANTS

Variants	Match Results				
	Set Node	*Node*	*Control Dependence*	*Data dependence*	*Eigen Value*
NetSky.ad	9	31	83	62	
NetSky.aa	6	23	61	42	0.70
NetSky.af	8	25	72	53	0.85
NetSky.c	8	22	68	51	0.81
NetSky.r	8	23	74	55	0.86
NetSky.t	8	26	71	55	0.87

In the experiment of NetSky, we firstly extracted features from NetSky.ad, and used it to detect other variants. The results are showed in Table 1. We find that the number of detected system calls is much more than matched system calls, which is caused by the junk system calls and the loops. For example, NetSky.ad used many *ReadFile* and *WriteFile* to copy itself, and we reduced these loops in abstracting process. The loop operation and equivalent operation replacing cannot impact on our method.

TABLE II. EIGEN VALUE OF SDBOT VARIANTS

Variants	Match Results				
	Set Node	*Node*	*Control Dependence*	*Data dependence*	*Eigen Value*
SdBot.b	10	14	103	61	
SdBot.up	10	14	103	61	1
SdBot.as	8	13	73	51	0.8
SdBot.m	10	14	65	47	0.79
SdBot.bx	10	13	76	50	0.83
SdBot.by	10	11	84	48	0.83

In the experiment of SdBot, we changed some code signatures by manual obfuscating to prove the effectiveness of our method, and the results are shown in Table 2. Taking

SdBot.bx e.g., we found many loops in it. Since our feature abstracting includes the process of reducing loops, these obfuscating methods cannot disturb our analysis.

To evaluate the false positive, we chose typical innocent software including Internet Explorer, FTP (File Transfer Protocol), Calculator and Notepad, and used the features abstracted from NetSky.ad and SdBot.b to computed the eigen value. The results were shown in Table 3. Obviously, the eigen values of innocent software are less than 0.7, which proves that the method in this paper can distinguish malware and innocent software well.

TABLE III. EIGEN VALUE OF SDBOT VARIANTS

Program	**Match Results**	
	NetSky.ad	**SdBot.b**
Notepad.exe	0.26	0.11
Calc.exe	0	0.16
Iexplore.exe	0.34	0.31
Ftpserv.exe	0.34	0.25

VII. RELATED WORK

A. Malwre analysis

Static analysis disassembles the binary code as first step, and then analyzes and abstracts the feature of malware. It can analyze the code all-around. One limitation is the dependence on disassemble technology; malware usually use obfuscation to prevent from being disassembled [7] and analyzed. Some works have been done to alleviate the influence of obfuscation, such as Christodorescu etc. [6] propose a method to treat code reordering, packing and junk-insertion, to recover the native code, but they didn't manage the equivalently replacing, etc.

Dynamic analysis analyzes the executing code while the code is running, so it is immune to most obfuscating technologies, but has difficulty in treating with multipath problem. Recently, Moser etc. [9] explore the multipath with help of system snapshot, which improve the performance of treating with multipath.

B. Feature extraction

Traditional characters of malware are signatures of sequences, such as Kirda etc. [2] described the characters with system call sequences, and Bailey etc.[4] used system message sequences to represent the characters. The sequence signatures deeply rely on the order of codes or behaviors, and are not resistance to obfuscation. Control flow graph (CFG) is another method to describe the features. It represents the malware behaviors as the execution process, and cannot deal with irreverent operations reordering. Recently works focus on describing malware with dependences, such as Christodorescu etc. [1] used system calls and the dependencies between them to describe features, and abstracted contrast sub graph between normal program and malware. Christodorescu etc. [3] relieve the interference from some obfuscation technologies and generate features through static method associated by semantic. This method focuses on instructions and performs weakly to deal with interference at behavior level.

VIII. CONCLUSIONS AND FUTURE WORK

This paper proposes a method to abstract the behavior characters of malware at semantic level. This method adopts dynamic taint analysis to abstract critical system calls and the dependencies between them to construct behavior dependence graphs. The behavior dependence graphs can describe the characters of malware well, and is robust to interference with help of semantic analysis. The evaluations on various malware show that our method is robust to obfuscation and can detect malware variant precisely.

In experiment, we find that the efficiency of analysis process is slightly low, which is inevitable because that the dynamic taint analysis need parse the instructions one by one. Another problem is that only one path can be analyzed in an execution, which is a common defect of dynamic analysis. Both of these problems are on our schedule. In future, we will also pay attention to the feature integrated from multi-sample.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China under Grant No. 61100228, the "Strategic Priority Research Program" of the Chinese Academy of Sciences under Grant No. XDA06030601 and XDA06010701, and the Project of IIE, Chinese Academy of Sciences, under Grant No.Y1Z0041101 and Y1Z0051101.

REFERENCES

[1] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), 2007.

[2] E. Kirda, C. Kruegel, G. Banks, G. Vigna, R. Kemmerer. Behavior-based spyware detection. In Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, August 2006.

[3] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant.. Semantics-aware malware detection. In IEEE Symposium on Security and Privacy, pages 32–46, 2005.

[4] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In Proceedings of the 10th Symposium on Recent Advances in Intrusion Detection (RAID'07), pages 178–197, 2007.

[5] Vugranam C. Sreedhar , Guang R. Gao , Yong-Fong Lee, Identifying loops using DJ graphs. ACM Transactions on Programming Languages and Systems (TOPLAS), v.18 n.6, pages 649-658, Nov. 1996.

[6] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, H. Veith. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, 2005.

[7] M. Christodorescu, S. Jha. Testing Malware Detectors. In Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004), pages 34–44, Boston, MA, July 2004.

[8] F. Bellard. Qemu, a fast and portable dynamic translator. In USENIX Annual Technical conference, FREENIX Track, April 2005.

[9] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In Proceedings of 2007 IEEE Symposium on Security and Privacy, 2007.

[10] VX Heavens. http://www.netlux.org